الجمهورية الجزائرية الديمقراطية الشعبية
People’s Democratic and Republic of Algeria

Ministry of Higher Education and Scientific Research
National School of Cyber Security

وزارة التعليم العالي والبحث العلمي
المدرسة الوطنية العليا في الأمن السيبراني

ALGORITHMS AND STATIC DATA STRUCTURES

CHAPTER 2: ALGORITHMIC FORMALISM (07h-08h)

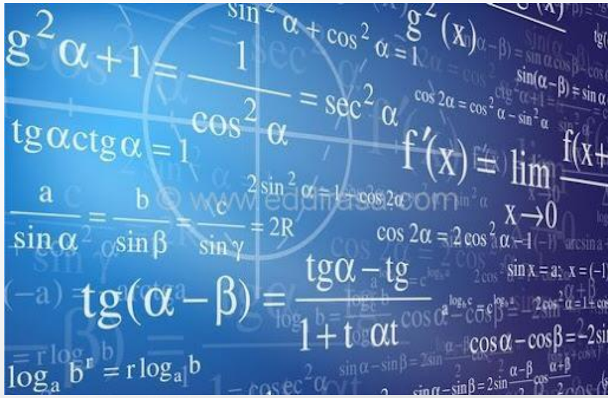
Prof. Abdelfetah HENTOUT
abdelfetah.hentout@enscs.edu.dz

23/10/2024

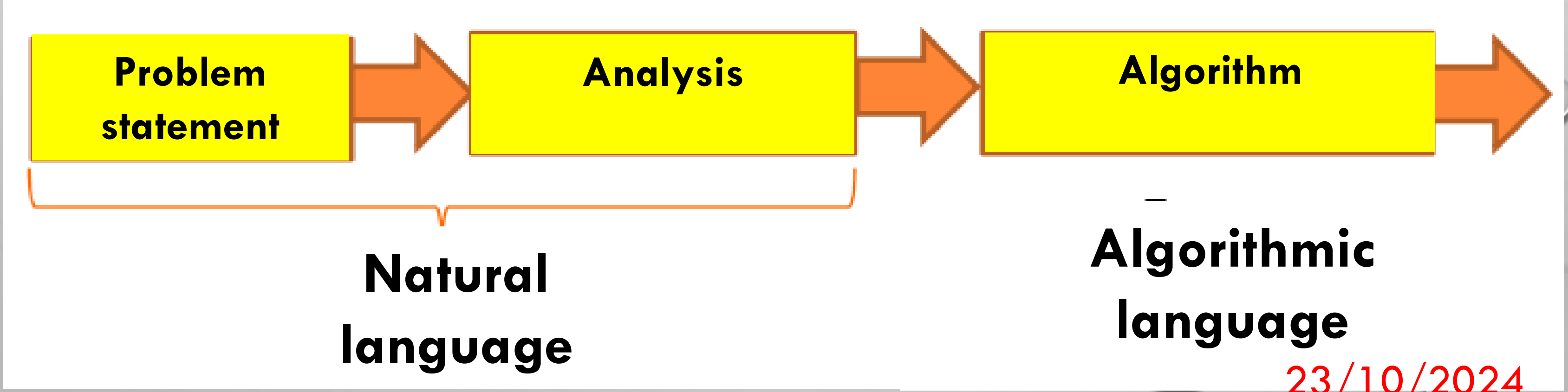
Algorithmic Formalism (1)

Why should we formalize?

- Once a problem has been analyzed, the designer must express the solution using a *universal formalism*, typically in the form of an *algorithm*.
- The challenge is to adopt a *common language*, allowing for the understanding of algorithms developed by others and vice-versa.



This highlights the importance of *formalism*



Algorithmic Formalism (2)

Definition (1)

- It is the precise representation of algorithms using formal language or notation, ensuring:
  - Clarity,
  - Unambiguity,
  - Systematic analysis for effective communication and verification.

Definition (2)

- An algorithmic formalism is a set of conventions (or rules) in which, is expressed, any solution to a problem.
- It is a common and precise language that allows communication without ambiguity.
  - It must be readable and understandable by multiple people.

Structure of an Algorithm (1)

An algorithm is organized (structured) into three parts

Header	**Algorithm** Name_of_the_Algorithm;
Declaration	**Environment** (Declaration of constants, variables, types, objects and modules used in the algorithm)

Body of the algorithm	**Begin**
	Action_1;
	Action_2;
	…
	Action_n;
	End;

Keywords:

- Algorithm
- Begin
- End

Structure of an Algorithm (2)

Header | **Algorithm** Name_of_the_Algorithm;

Header

- It simply serves to identify the algorithm.
- Indicates the name of the algorithm and possibly a brief description of its purpose.
- Generally, a descriptive name should be given to allow the reader to understand what the algorithm will do.

Algorithm Name_of_the_Algorithm; // role of the algorithm (optional)

Examples of Headers

- **Algorithm** Addition;
- **Algorithm** GCD;
- **Algorithm** Divisors;
- **Algorithm** Calculate_Circle_Area;
- **Algorithm** Sum_Integers;

Structure of an Algorithm (3)

Declaration

- **Constants**
- **Types**
- **Variables**
- **Sub-programs (functions, procedures)**

Declaration:

- Declares the necessary variables:
  - The identifiers within their data types
  - The input parameters.
- Declarations of input data, meaning the elements that are essential for its proper functioning.
- Declarations of output data, meaning the elements calculated or produced by the algorithm.
- Declarations of local data that are essential to the algorithm.

Structure of an Algorithm (4)

Environment (declaration part)

- It contains a comprehensive list of the objects used and manipulated within the body of the algorithm.
- It is placed at the beginning of the algorithm.
- Every algorithm uses objects that are declared in its environment.
- For each object, it is necessary to correspond:
  - A NAME (Identifier) that allows it to be designated and distinguished from the other elements.
  - A TYPE that indicates the nature of the set in which the object takes its values.
  - A VALUE assigned to this object at a given moment.

Variables **Identifier: Type;**

Structure of an Algorithm (5)

Identifier

- The construction of a name to uniquely designate an object follows precise rules in algorithmics and programming.
- The chosen name is called an **Identifier.**

Rules to follow:

- An IDENTIFIER must begin with an uppercase/lowercase LETTER and can consist of letters, digits, and/or the symbol "_" (underscore).
- An IDENTIFIER must not contain spaces (whitespace characters), accented characters, or characters such as "%", "?", "*", ".", "-" , …

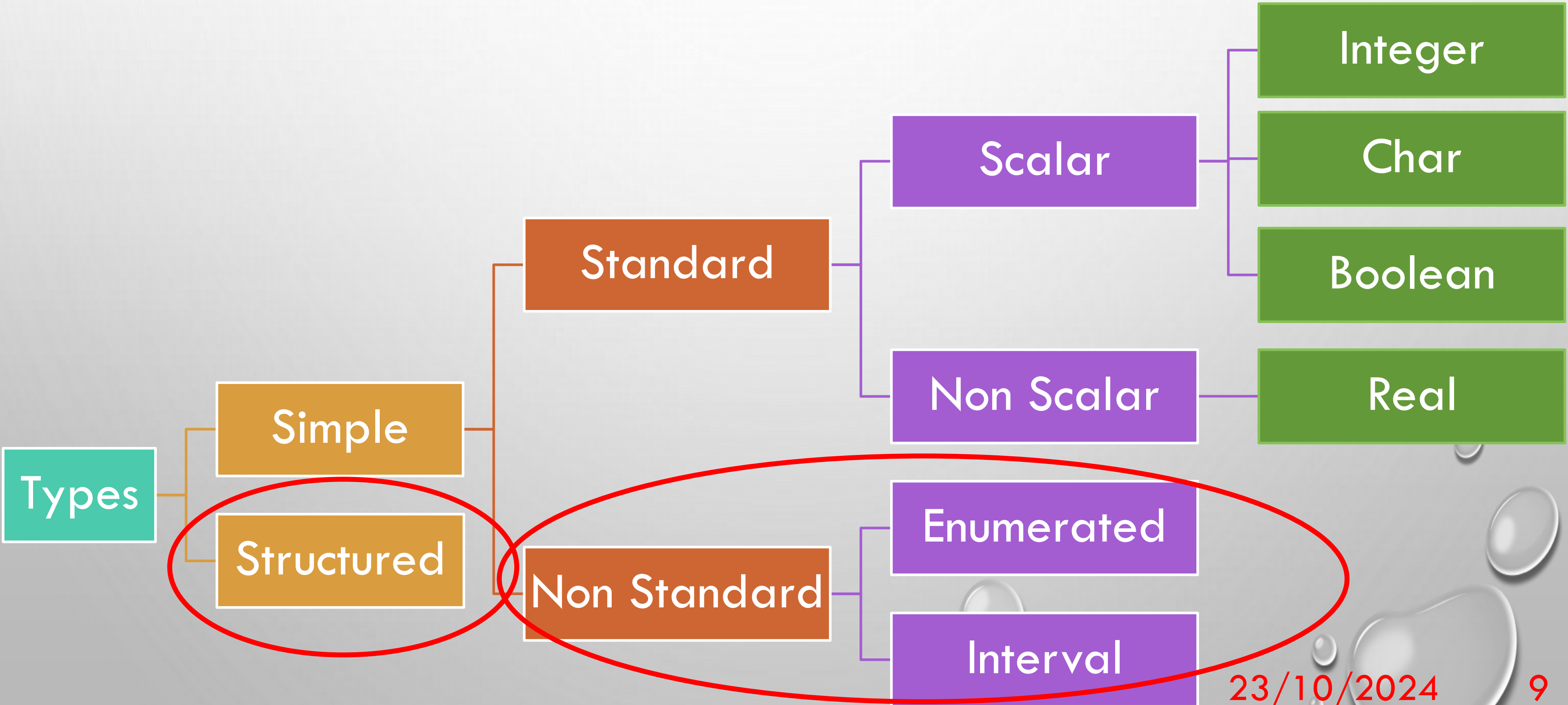
Examples of incorrect identifiers

- 34x (it does not start with a letter).
- velocity object (it contains a space between velocity and object).
- engine450.7hp! (it contains a "." and a "!").

Structure of an Algorithm (6)

TYPE

- It represents:
  - The set of values that an object belonging to it may take.
  - The authorized operations on that object.



Structure of an Algorithm (7)

Integer

- This is the set of integers
- In mathematics, this set is infinite
- On a computer, the values are limited by the length of the machine words.

Type	Interval	Length
shortint	-128...127	01 byte (08bits)
integer	-32 768...32 767	02 bytes
longint	-2 147 483 648...2 147 483 647	04 bytes
byte	0...255	01 byte
word	0...65 536	02 bytes

Structure of an Algorithm (8)

Char (Character)

- It corresponds to a single character.
- It includes all alphanumeric characters (alphabetic, numeric, special characters and space).
- The character set can vary.

- The values are ordered according to the internal character code:
  - ASCII (American Standard Code for Information Interchange)

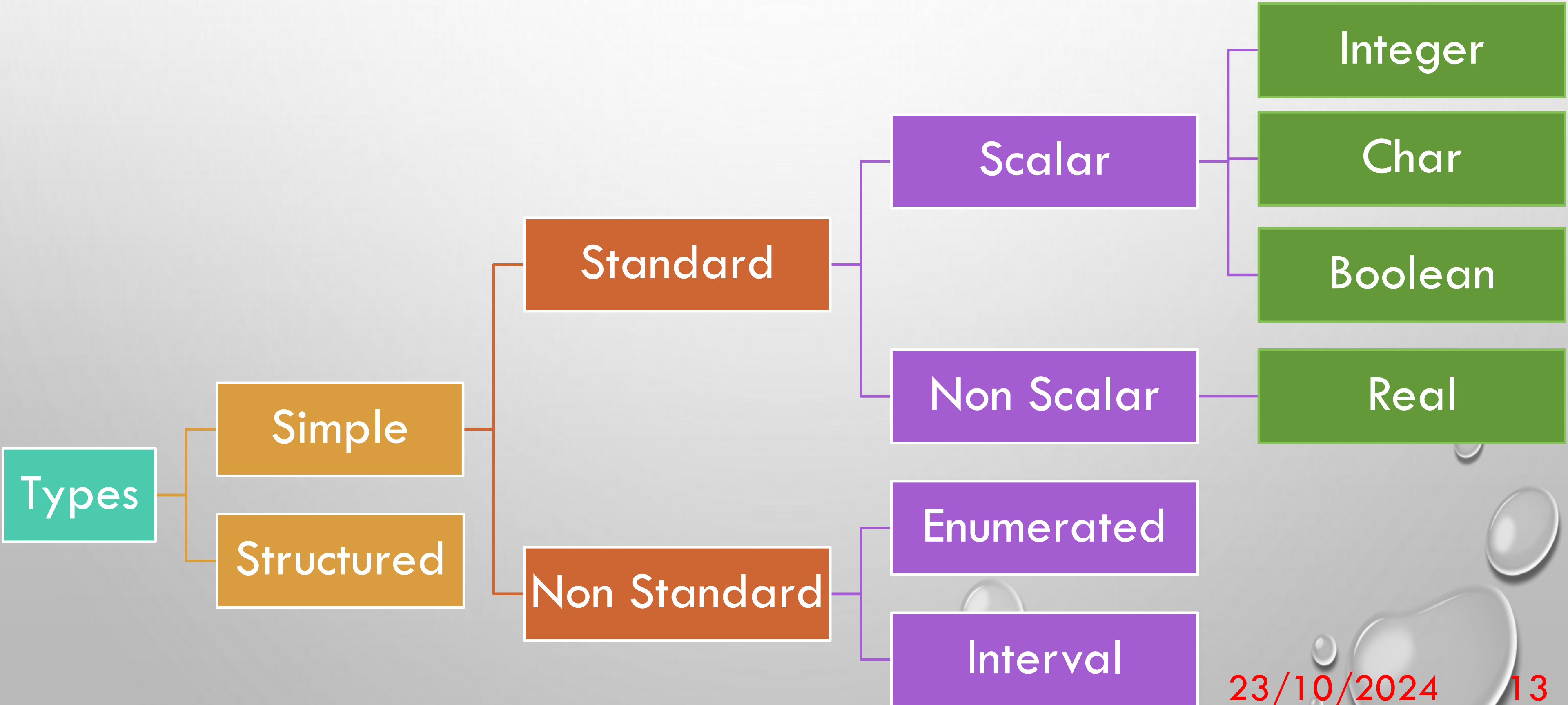
Structure of an Algorithm (9)

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32		48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	‘	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	

Structure of an Algorithm (10)

Boolean (Logical)

- This is the set of two values:
  - True
  - False



Structure of an Algorithm (11)

Real

- This is the set of numbers having a fractional part.
- This set is also limited
- The limits are wider and depend on the internal representation (Architecture of computers).

±integer part.decimal part

Type	Interval	Length	Number of digits
Real	$2.9*10^{-39}...1.7*10^{38}$	06 bytes	11-12
Single	$1.5*10^{-45}...3.4*10^{38}$	04 bytes	07-08
Double	$5.8*10^{-324}...1.7*10^{308}$	08 bytes	15-16

Structure of an Algorithm (12)

Constant

- Some identifiers have a constant value (that does not change) during the **entire** execution of the algorithm.
- This data is not variable but constant.
- These identifiers are called constants.
- They are declared at the beginning of the algorithm by following the identifier with its value.

Constant Constant_Identifier=Value;

How does it work ?

- Some information manipulated by a program never changes.
- Instead of explicitly placing their value in the text of the program, it is preferable to give them a symbolic (and meaningful) name.
- This is the case for the value of:
  - π (Constant of Archimedes): **PI=3.14159265359**
  - Gravity on earth: **G=9.80665m/s²**
  - etc.

Structure of an Algorithm (13)

Body

- The body of the algorithm contains a sequence of:
  - Actions (instructions).
  - Executed in the order they appear in the algorithm.
- It starts by **Begin**
- It ends by **End;**

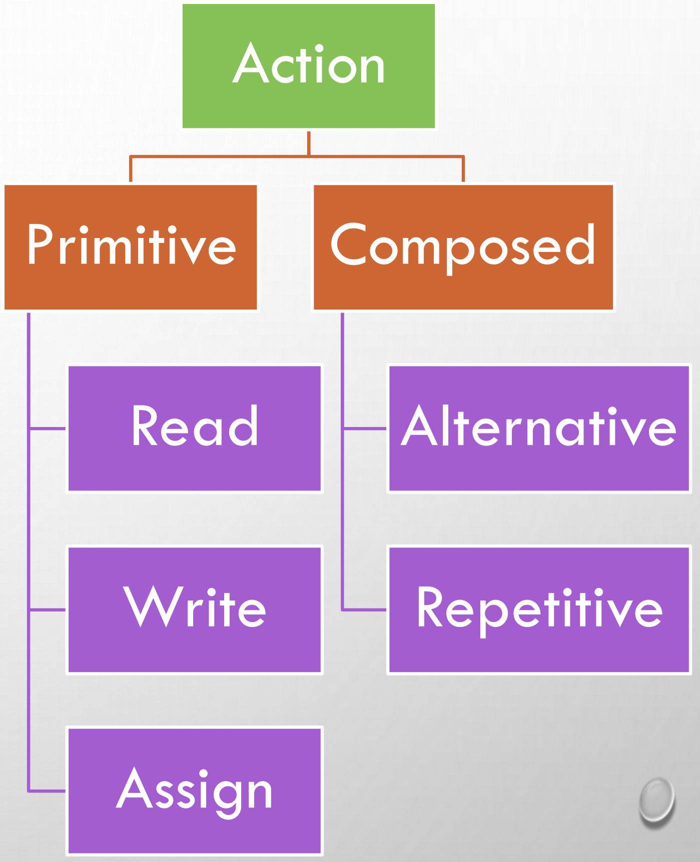
Body of the algorithm

Begin
Action_1;
Action_2;
…
Action_n;
End;

Begin
read(a);
read(b);
average=(a+b)/2;
print(average);

Action

- An event that acts on the variables by modifying their values or by observing them.
- Two types of actions:
  - o **Elementary (primitive)**:
    - ▪ Executed without any additional information.
  - o **Composed**: Composed of several other primitive actions.
    - ▪ Executed under certain conditions (once or repeated).
    - ▪ Control structures.



Primitive actions (1)

Reading

- Reading a value from an input device (usually the keyboard) and then storing the read value in a variable.
- In case of several variables to read, simply separate them with ",".

read(variable);

read(variable_1, variable_2 … variable_n);

- Reading blocks the execution until the values are received and stored in the variables.
- The values read from the keyboard are assigned to the variables respectively, while verifying type compatibility.

```
a, b: integer;
delta: real;
read(a);
read(b);
read(delta);
```

Primitive actions (2)

Writing/Displaying

- Write/Display the values of the objects mentioned in the list.
- In case of several variables to write/display, simply separate them with ",".

print(variable);
print(variable_1, variable_2 … variable_n);
print("The value of Expression = ", expression);

An object can be:

- A value, such as "5", "Hello" …
  - A value is displayed as it is.
- A variable, such as x, y …
  - A variable is replaced by its value.
- An **expression**, such as (x+y)/2, (a **or** b),√delta …
  - First, **expression** is evaluated.
  - Second, result is displayed.

Primitive actions (3)

Example: Give the results of each action

- a=5; print (a);
- c=4; print ("c=", c) ;
- b=a*a-c; print (a*a-c);
- print(a, b, c);
- print(a,   b,   c);
- print(a, " ", b, " ", c);
- print("a=", a, " b=", b, " c=", c);

- 5
- c=4
- 21
- 5214
- 5214
- 5 21 4
- a=5 b=21 c=4

Primitive actions (4)

Assignment

- Evaluate the entity located on the right of the assignment (expression).
- Put the result in the entity located on the left.

- Expression type must be compatible with the type of the left variable.
- The previous value of the left variable is lost (overwritten).

Instruction	y	x	x1	y1
y=2;	2	?	?	?
x=y;	2	2	?	?
x=x+2;	2	4	?	?
x1=y+x;	2	4	6	?
y1=x1+2;	2	4	6	8
y1=y1*2;	2	4	6	16
x=y1+3;	2	19	6	16

Primitive actions (5)

```
Algorithm Second_Degree_Solution;
Variables
    a, b, c, delta: integer;
    x1, x2: real;
Begin
    read(a);
    read(b);
    read(c);                    // ax²+bx+c=0
    delta=b*b-4*a*c;            // delta≥0
    x1=(-b-√delta)/(2*a);          // √delta returns the square root of delta
    x2=(-b+√delta)/(2*a);
    print(delta);
    print(x1);
    print(x2);
End;
```

Conditional (1)

IF statement

```
IF (Condition)
Begin
    Action_1;
    …                //BLOCK
    Action_n;
End;
Action_(n+1);
```

- If **Condition** is TRUE, the BLOCK (*Action_1 … Action_n*) is executed.
- Otherwise, it goes to *Action(n+1)*.

- **Condition** is a Boolean expression:
  - An expression that evaluates to either **TRUE** or **FALSE**.

Important

- A block is a coherent set composed of one or several actions.
- A block begins with **Begin** and ends by **End;**
- In case a block is composed of one primitive action only, **Begin** and **End** are facultative.

IF statement

```
Algorithm Second_Degree_Solution;
Variables
    a, b, c: integer;     delta, x1, x2: real;
Begin
    read(a);    read(b);    read(c);     // ax²+bx+c=0
    delta=b*b-4*a*c;
    IF(delta>=0)
    Begin
        x1=(-b-√delta)/(2*a);
        x2=(-b+√delta)/(2*a);
        print(x1);
        print(x2);
    End;
End;
```

Alternative (1)

IF-ELSE statement

```
IF (Condition)
Begin
     Action_1;
     …                //BLOCK 1
     Action_3;
End;
Else
Begin
     Action_4;
     …                //BLOCK 2
     Action_8;
End;
Action_9;
```

- If **Condition** is TRUE, the BLOCK1 (*Action_1* … *Action_3*) is executed.
- Otherwise, the BLOCK 2 (*Action_4* … *Action_8*) is executed.

Alternative (2)

IF-ELSE statement

```
Algorithm Sign_Integer;
Variables
    number: integer;
Begin
    read(number);
    IF(number<0) print("Negative number");
    Else          // number≥0
    Begin
        IF(number>0) print("Positive number");
        Else    print("Zero");        // number=0
    End;
End;
```

Alternative (3)

IF-ELSE statement

```
Algorithm Second_Degree_Solution; // Find solutions in ℜ
Variables  a, b, c, delta: integer;  x1, x2: real;
Begin
    read(a);    read(b);    read(c);     // ax²+bx+c=0
    delta=b*b-4*a*c;
    IF(delta<0) print("No solutions");
    Else          // delta≥0
    Begin
        If(delta>0)
        Begin
            x1=(-b-√delta)/(2*a);          // SQRT(delta), Square Root
            x2=(-b+√delta)/(2*a);
        End;
        Else    x1=x2=-b/(2*a);
        print(x1);       print(x2);
```

Alternative (4)

SWITCH-CASE statement

```
Algorithm Switch_Case;
Variables   a: integer;
Begin
    read(a);
    IF (a=1)
    Begin
        Block 1;
    End;
    Else
        IF (a=2)
        Begin
            Block 2;
        End;
        Else
        Begin
            IF (a=3)
            Begin
                Block 3;
            End;
            Else
                IF (a=4)
                Begin
                    Block 4;
                End;
                Else
                Begin
                    Block 5;
                End;
            End;
        End;
    End;
End;
```

Alternative (5)

SWITCH-CASE statement

```
switch (expression)
Begin
    case value1:
        Action_1;
        ...
        Action_6;   break;
    case value2:
        Action_7;
        ...
        Action_10; break;
    ....
    default:
        Action_n;
        break;
End;
```

- **switch case** statement is an alternative to the **IF-ELSE** statement.
  - It is used to execute the conditional code based on the value of the **expression** specified in the switch statement.
- The **default** block is optional.
  - It defines the actions to be executed if the value of **expression** does not correspond to any case.

Note

- expression should evaluate to either integer or character.
- It cannot evaluate any other data type.

Alternative (6)

SWITCH-CASE statement

```
Algorithm Switch_Case;
Variables
    a: integer;
Begin
    read(a);
    switch (a)
    Begin
        case 1:
            Block 1;
            break;
        case 2:
            Block 2;
            break;
        case 3:
            Block 3;
            break;
        case 4:
            Block 4;
            break;
        default:
            Block 5;
            break;
    End;
End;
```

Alternative (7)

SWITCH-CASE statement

```
Algorithm switch_case_statement;
Variables cas: integer;
Begin
    read(cas);
    switch (cas)
    Begin
        case 1:
            printf("Case 1 is executed");   break;
        case 2:
            printf("Case 2 is executed");   break;
        default:
            printf("Default Case is executed");     break;
    End;
End;
```

Alternative (8)

Example: Write an algorithm that reads the number of a month and displays the number of days.

```
Algorithm number_days_months_If-Else;
Variables month : Integer;
Begin
    print("Give the number of the month [1…12]:");
    read(month);
    IF ((month=1) or (month=3) or (month=5) or (month=7) or (month=8)
    or (month=10) or (month=12))      print("This month is 31 days");
    Else
        IF ((month=4) or (month=6) or (month=9) or (month=11))
            print("This month is 30 days");
        Else
            IF (month=2) print("This month is 28 or 29 days");
            Else print ("ERROR: Invalid month");
End;
```

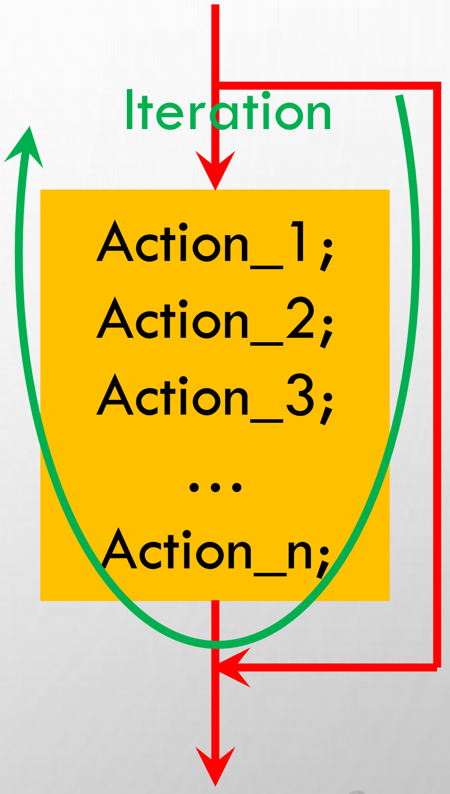
Alternative (8)

Example: Write an algorithm that reads a month number and displays the number of days.

```
Algorithm number_days_months_Switch_Case;
Variables month : Integer;
Begin
    print("Give a number of a month [1…12]:");  read(month);
    switch (month)
    Begin
        case 1, 3, 5, 7, 8, 10, 12: print("This month is 31 days");
                                    break;
        case 4, 6, 9, 11:           print("This month is 30 days");
                                    break;
        case 2:                     print(" This month is 28 or 29 days");
                                    break;
        default:                    print ("ERROR: invalid month"); break;
    End;
End;
```

Loops

- Many processes require repeating a block several times, with possible variation of parameters.
- A Loop allows executing an instructions block several times according to a condition.
- Each execution of the block is called an **iteration.**
- There are three forms of repetitive (loops):
  - **For.**
  - **While.**
  - **Do … While.**
- When the number of iterations is a priori known, often use **For** loop.
- Otherwise (unknown), use the other loops (**While, Do … While**).
  - If the loop is executed at least once (≥1), use **Do … While** loop.
  - If the loop may not be executed (≥0), use **While** loop.



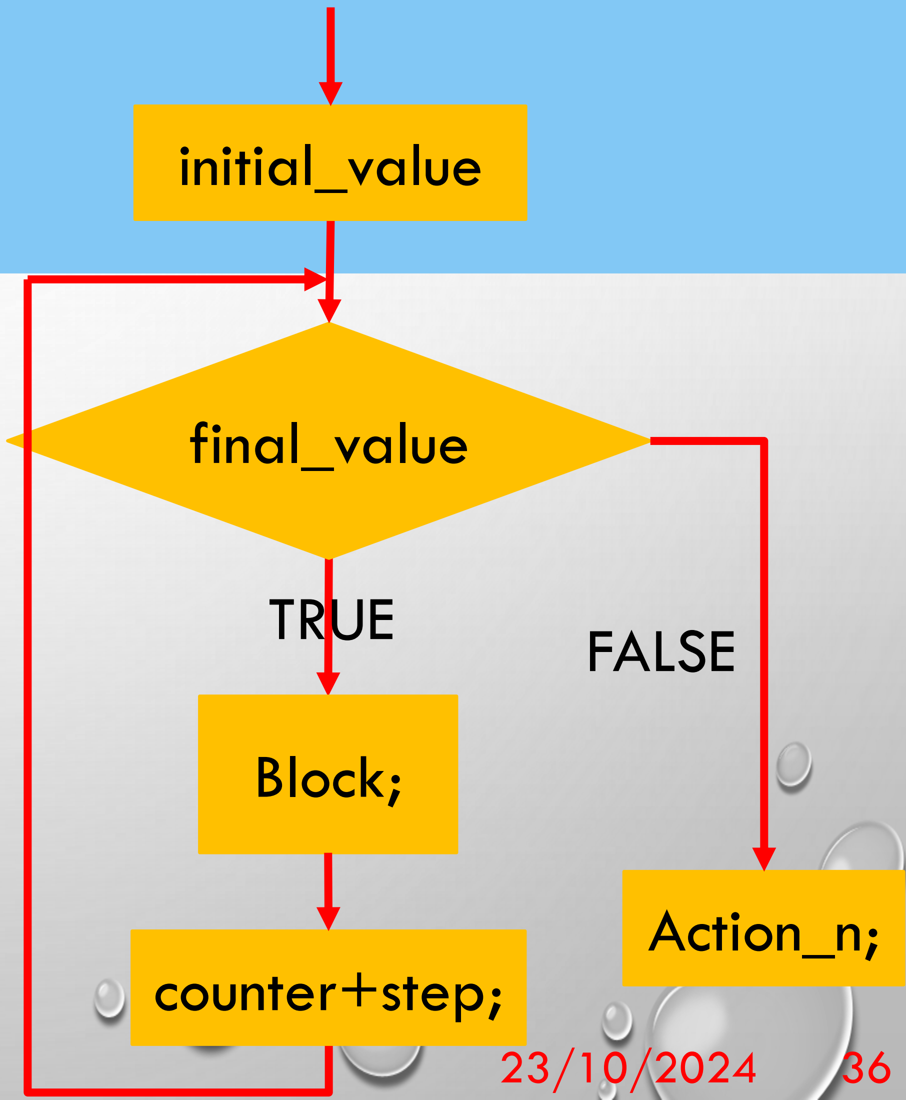
FOR loop (1)

For (*counter*=initial_value; *counter*≤final_value; *step*)
Begin
    Block;
End;
Action_n;

- *counter* is used to enumerate the iterations.
  - *counter* is called the loop counter.
- *initial_value*, *final_value* and *step*: can be integer constants, integer variables, or integer expressions.
- **For** loop is used if:
  - repetitions number is known.
  - *initial_value* and *final_value* of *counter* are known.
- *counter* will take the values [*initial_value* … *final_value*] by adding *step* value each time.
- For each value of *counter*, the block is executed.

FOR loop (2)

For (*counter*=initial_value; *counter*≤final_value; *step*)
Begin
    Block;
End;
Action_n;



Warning!

- Do not modify the *counter* and the *final_value* inside the loop:
  - disrupts the number of iterations planned by **For** loop.
  - presents the risk of an infinite loop.

FOR loop (3)

For (*counter*=initial_value; *counter*≤final_value; *step*)
Begin
    Block;
End;
Action_n;

How does it work?

1. Initialize *counter*=initial_value;
2. Evaluate the stop ***Condition***: (counter>final_value)?
3. In case (***Condition* = TRUE**)
   - Execute Block;
   - counter=counter+step;
   - Goto 2;
4. Otherwise (**Condition = FALSE**)
   - Execute Action_n;

FOR loop (4)

Example 1: Write three algorithms to display the integer numbers multiple of 10 in the interval [0, 50]

```
For (i=0; i≤50; i=i+10)
Begin
    print(i);
End;
```

```
0
10
20
30
40
50
```

FOR loop (5)

Example 2: Write three algorithms to display the integer numbers multiple of 5 in the interval [30, 5]

```
For (i=30; i≥5; i=i-5)
Begin
    print(i);
End;
```

```
30
25
20
15
10
5
```

FOR loop (6)

Example 3: Write an algorithm to display the sum of integer numbers [1, 100]

```
Algorithm sum_100_integers;
Variables
    i, sum: integer;
Begin
    sum=0;
    For (i=1; i≤100; i=i+1)
    Begin
        sum=sum+i;
    End;
    print("Sum of the first 100 numbers=", sum);
End;
```

FOR loop (7)

Example 4: Write an algorithm to display the integer numbers [1, 100] multiple of 5.

```
Algorithm multiple_5_less100_1st;
Variables
    i: integer;
Begin
    For (i=5; i≤100; i=i+5)
    Begin
        print(i);
    End;
End;
```

```
Algorithm multiple_5_less100_2nd;
Variables
    i: integer;
Begin
    For (i=1; i≤20; i=i+1)
    Begin
        print(5*i);
    End;
End;
```

FOR loop (8)

Example 5: Write an algorithm to display all the divisors of an integer number *n*.

```
Algorithm divisors_of_n;
Variables n, i: integer;
Begin
    print("Give a positive integer number, n=");
    read(n);
    print("The divisors are: 1, ");
    For (i=2; i≤n/2; i=i+1)
    Begin
        If(n%i==0) print(i, ", ");
    End;
    print(n);
End;
```

FOR loop (9)

Nested FOR loops

```
For (counter_1=initial_value_1; counter_1≤final_value_1; step_1)
Begin
    Action_1 … Action_m;
    For (counter_2=initial_value_2; counter2≤final_value_2; step_2)
    Begin
        Block_of_Inner_Loop;
    End;
    Block_of_Outer_Loop;
End;
```

- A nested loop has one loop inside of another.
- When a loop is nested inside another loop, the inner loop runs many times inside the outer loop.
- Each iteration of the outer loop, the inner loop will be re-started.
- The inner loop must finish all of its iterations before the outer loop can continue to its next iteration.

FOR loop (10)

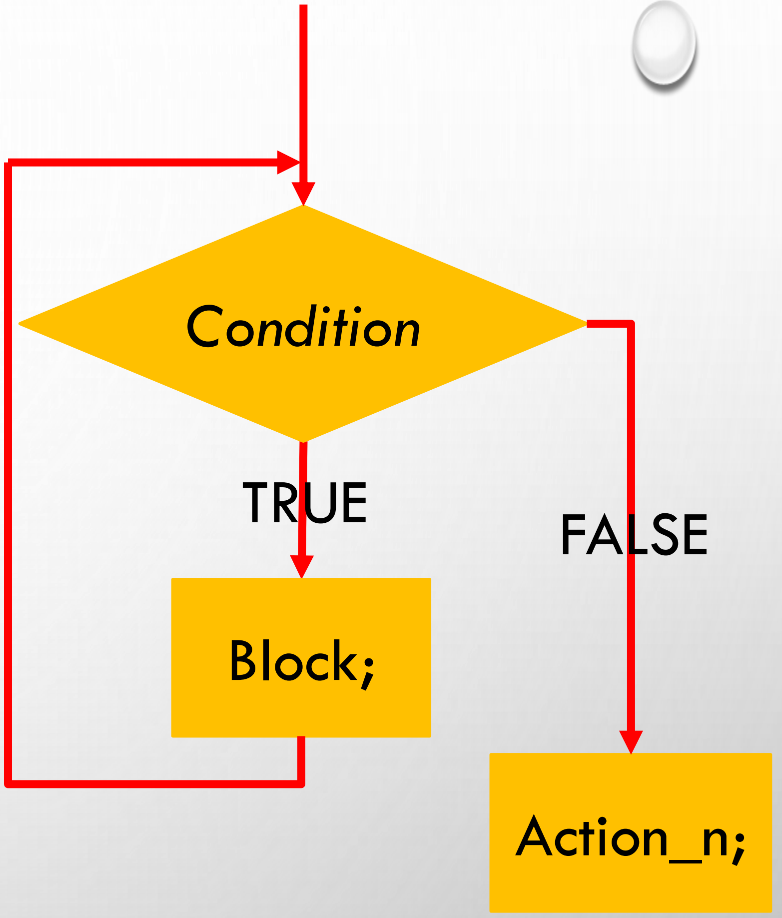
Example 6: Write an algorithm to display the elements (rows and columns) of a matrix [3,2] of two dimensions.

```
Algorithm Rows_Columns_2D_Matrix;
Variables row, column: integer;
Begin
    For (row=1; row≤3; row=row+1)
    Begin
        For (column=1; column≤2; column=column+1)
         Begin
            print("Element[", row, ",", column]"=", matrix[row, column]);
        End;
    End;
End;
```

```
Element[1,1]=xy
Element[1,2]=xz
Element[2,1]=yx
Element[2,2]=yz
Element[3,1]=zx
Element[3,2]=zy
```

WHILE loop (1)

While (*Condition*)
Begin
    Block;
End;
Action_n;



The form of the **While(Condition)** loop is as follows:

- ***Condition*** must be computable (before each iteration).
- **Block** of the loop may never be executed.
- **Block** must modify the condition variables (to avoid infinite loop).

WHILE loop (2)

```
While (Condition)
Begin
    Block;
End;
Action_n;
```

How does it work?

1. Evaluate the **Condition**?
2. In case (**Condition = TRUE**)
   - Execute Block;
   - **Goto** 1;
3. Otherwise (**Condition = FALSE**)
   - Execute Action_n;

WHILE loop (3)

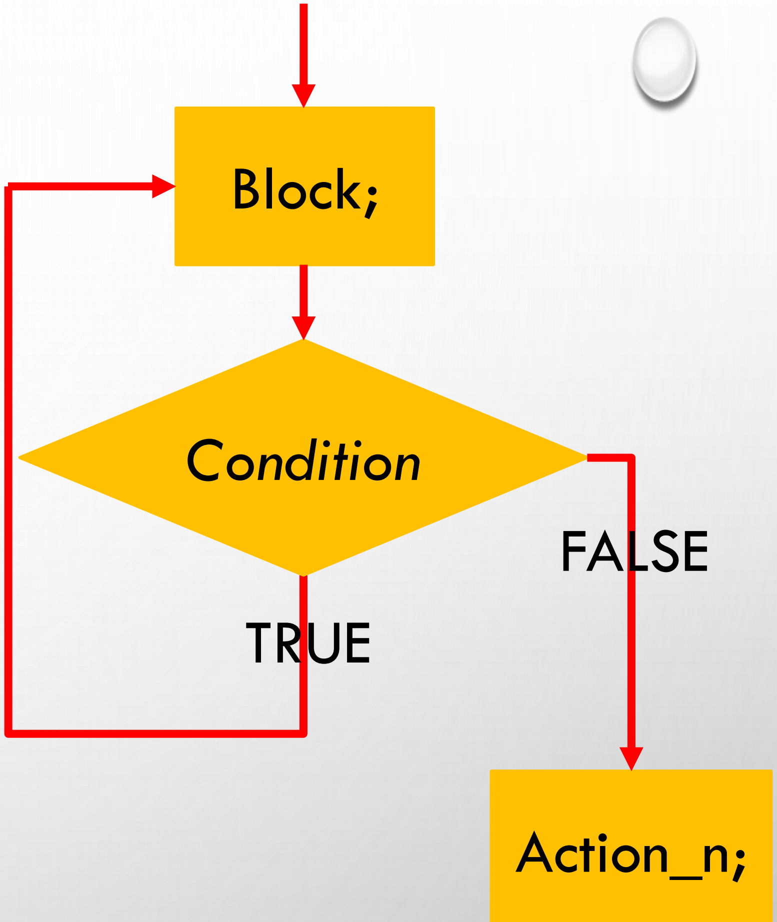
Example 1: Write an algorithm to calculate the set of divisors of a number *n*.

```
Algorithm Number_Divisons;
Variables n, i: integer;
Begin
    print("Enter a number, n=");    // read a number n
    read(n);
    i=1;
    While(i≤n)        // run i = 1, 2 … n
    Begin
        If (n%i==0)            // i is a divisor of n
            print(i);
        i=i+1;                 // Make sure we move to the next number
    End;
End;
```

Do … WHILE loop (1)

Do
Begin
    Block;
End;
While(*Condition*);
Action_n;

Do
    Block;
While(*Condition*);
Action_n;



- Both forms are equivalent:
    - **Begin-End;** are optional.
- *Condition* is calculated after each iteration.
- *Block* of the loop is executed at least once.
- *Block* must modify the variables of the condition (to avoid infinite loop).

Do … WHILE loop (2)

```
Do
Begin
    Block;
End;
While(Condition);
Action_n;
```

```
Do
    Block;
While(Condition);
Action_n;
```

How does it work?

1. Execute Block;
2. Evaluate the **Condition**?
3. In case (**Condition = TRUE**)
   - **Goto** 1;
4. Otherwise (**Condition = FALSE**)
   - Execute Action_n;

Loops (1)

Example 1: Translate the following **"For"** loop into **"While"** loop.

```
For (i=0; i≤50; i=i+10)
Begin
    print(i);
End;
```

```
i=0;
While(i≤50)
Begin
    print(i);
    i=i+10;
End;
```

Loops (2)

Example 2: Translate the following **"For"** loop into **"While"** loop.

```
For (j=30; j≤5; j=j-5)
Begin
    print(j);
End;
```

```
j=30;
While(i≥5)
Begin
    print(j);
    j=j-5;
End;
```

Loops (3)

Example 3: Translate the following **"For"** loop into **"While"** loop.

```
For (i=1; i≤3; i=i+1)
Begin
    For (j=1; j≤2; j=j+1)
    Begin
        print("i=", i, " j=", j);
    End;
End;
```

```
i=1;
While (i≤3)
Begin
    j=1;
    While (j≤2)
    Begin
        print("i=", i, " j=", j);
        j=j+1;
    End;
    i=i+1;
End;
```

Loops (4)

Example 4: Translate the following **"For"** loop into **"Do…While"** loop.

```
For (i=0; i≤50; i=i+10)
Begin
    print(i);
End;
```

```
i=0;
Do
Begin
    print(i);
    i=i+10;
End;
While(i≤50)
```

```
i=0;
Do
    print(i);
    i=i+10;
While(i≤50)
```

Loops (5)

Example 5: Translate the following **"For"** loop into **"Do…While"** loop.

```
For (j=30; j≤5; j=j-5)
Begin
    print(j);
End;
```

```
j=30;
Do
Begin
    print(j);
    j=j-5;
End;
While(j≥5)
```

```
j=30;
Do
    print(j);
    j=j-5;
While(j≥5)
```

Loops (6)

Example 6: Translate the following **"For"** loop into **"Do…While"** loop.

```
For (i=1; i≤3; i=i+1)
Begin
    For (j=1; j≤2; j=j+1)
    Begin
        print("i=", i, " j=", j);
    End;
End;
```

```
i=1;
Do
    j=1;
    Do
        print("i=", i, " j=", j);
        j=j+1;
    While (j≤2);
    i=i+1;
While (i≤3);
```

```
i=1;
Do
Begin
    j=1;
    Do
    Begin
        print("i=", i, " j=", j);
        j=j+1;
    End;
    While (j≤2);
    i=i+1;
End;
While (i≤3);
```

Loops (7)

Remarks

- **While** and **Do … While** forms are used when the number of loops (iterations) is unknown:
  - For **Do … While** form, it is at least equal to 1.
  - For **While** form, it can be equal to 0.

- This difference often allows to choose between both forms.

Warning!

Beware of design errors that can lead to an "infinite loop", when **Condition** always remains verified.

Loops (8)

Exercise 1: Write an algorithm that reads a positive integer *n*, then double it as needed until it exceeds 60.

```
Algorithm Double_Less_60;
Variables n : Integer;
Begin
    Do
        print("Give a positive integer n=");
        read(n);
    While(n≤0);
    While (n≤60)
    Begin
        n=n*2;
    End;
    print("n=" , n);
End;
```

Exercise 2: Write an algorithm that calculates the sum of *n* integers (*n*>0) as well as their average.

```
Algorithm Sum_Average;
Variables number_integers, x, sum : Integer; moy: real;
Begin
    Do
        print("Give a positive integer n=");    read(number_integers);
    While(n≤0);
    For (i=1; i≤number_integers; i=i+1)
    Begin
        print("Give an integer x=");    read(x);
        sum=sum+x;
    End;
    moy=sum/number_integers;
    print("Sum=", sum);     print("Average=", moy);
End;
```

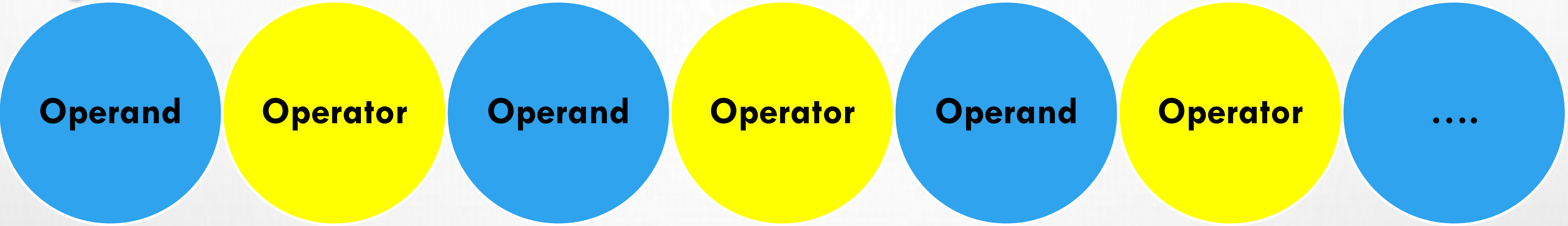
Loops (10)

Exercise 3: Write an algorithm that displays the minimum of *n* read integer numbers (*n*>0)

```
Algorithm min_N_Integers;
Variables n, i, min: Integer;
Begin
    Do
        print("Give a positive integer n=");    read(n);
    While (n≤0);
    print("Give an integer=");      read(min);
    For (i=2; i≤n; i=i+1)
    Begin
        print("Give and integer x=");      read(x);
        If(x<min) min=x;
    End;
    print ("Minimum=", min);
End;
```

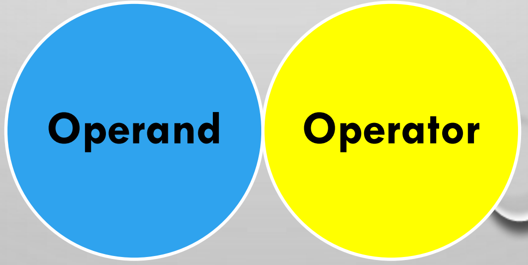
Expressions

Expressions



Binary operators

Unary operators



Arithmetic expressions (1)

- **Operand**: integers, reals
- **Operators:** + (plus), - (minus), / (slash), * (asterisk), % (modulo)

- **/ (slash): division:**
  - **result=operand_1/operand_2;**
  - In case **result** is integer, the division result will be an integer.
  - In case **result** is real, the division result will be an integer.

```
Variables  ope1, ope2, res_int: integer;
           res_real: real;
Begin
    read(ope1);    read(ope2);
    res_int=ope1/ope2;
    res_real=ope1/ope2;
    print("res_int=", res_int);
    print("res_real=", res_real);
End;
```

```
ope1=20;
ope2=3;
res_int=6;
res_real=6.33;
```

Arithmetic expressions (2)

- **Operand**: integers, reals
- **Operators:** + (plus), - (minus), / (slash), * (asterisk), % (modulo)

- **% (modulo): division remainder:**
  - **remainder=operand_1 % operand_2;**

```
Variables  ope1, ope2, rem: integer;
Begin
    read(ope1);
    read(ope2);
    rem=ope1 % ope2;
    print("remainder=", rem);
End;
```

ope1=20;
ope2=3;
remainder=2;

Arithmetic expressions (3)

Priority of operators:

*, /, %

+, -

- The hierarchy of operators defines how an expression is evaluated.
  - It starts with operators with highest hierarchy.
  - Then, moves to those with immediately lower hierarchy.
- In case of arithmetic expression:
  - It starts by carrying out all multiplications, divisions, and modulos.
  - Then, it moves to additions and subtractions.
- In case of same hierarchy, expression is evaluated from left to right.

v = a + b / d - x * z + x * y

Arithmetic expressions (4)

- Types must be respected; otherwise, errors may occur during execution.
- Type of arithmetic expression results depends on operators and operands types:

Operators	Operands type	Result type
+, -, *	integer … integer real … real real … integer integer … real	integer real real real
/ (real variable)	integer / integer real / real real / integer integer / real	real real real real
/ (integer variable)	integer / integer real / real real / integer integer / real	integer integer integer integer
%	integer % integer	integer

Logical expressions

- **Operand:** TRUE, FALSE
- **Operators:** && (and), | | (or), ! (not)

Priority of operators:

!

&&

| |

- **&&** and **| |** operators are binary because they require two operands.
- **!** operator is unary because it requires only one operand.

result = a | | c && b && ! d

Relational expressions

- **Operands:** Numerical, Alphanumerical
- **Operators:** =, >, >=, <, <=, !=

Priority of operators:

- Operators have the same priority.

a<b

a!=b

a>=b

Mixed expressions

- **Operands:** any
- **Operators:** any

Priority of operators:

!

*,   /,   %,   &&

+,   -, ||

=,   >,   >=,   <,   <=,   !=

(a * b > c) || (a / c = 0) && (x || ! y) && (b % c >= r)

Utilization of parentheses (1)

- Complex expressions require using parentheses, because they must be expressed in a linear form (on the same line).
- Expressions inside parentheses are evaluated first, starting with the innermost ones.

Example 1

$$Value = \frac{L * B * F}{\frac{F * B + n}{d} + e}$$

Value = (L* B* F) /((((F * B) + n) / d) + e)

Value = L * B * F /((F * B + n) / d + e)

Utilization of parentheses (2)

Example 2

$$frac = \frac{a+b}{c-d}$$

frac=(a+b)/(c-d)

Example 3

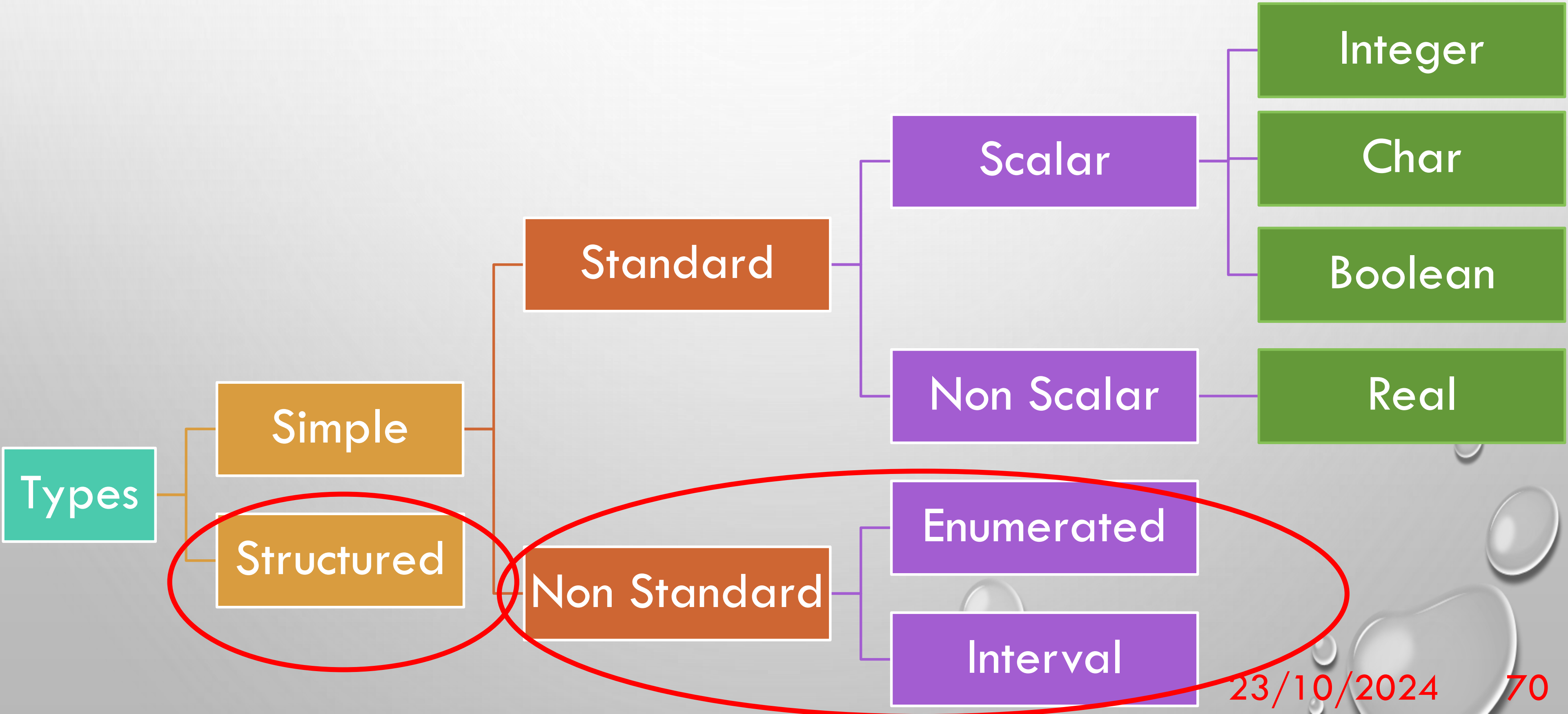
$$data = (\frac{percentage}{15} - \frac{var1 + var2}{20}) * (\mathrm{var3} + \mathrm{var4})$$

data= ((percentage/15) – ((var1+var2) / 20))*(var3+var4)

Non-standard types (1)

Specific types may be defined to the problem through non-standard types:

- Enumerated.
- Interval.



Non-standard types (2)

Enumerated type

- It defines an ordered set of values designated by identifiers (constants) (256 max).

Type *name_of_the_type={element_1, element_2 … element_n};*

Examples

Type

- Days={Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
- Colors={Red, Green, Blue, Yellow, Orange, White, Black};
- Size={Very big, Big, Average, Small, Very small};
- Answers={Yes, No, May_be};
- Feast={Aid_El_Fitr, Aid_El_Adha, Moharram, Achoura, Mouloud};
- Months={January, February, March, April, May, June, July, August, September, October, November, December};

Non-standard types (3)

Interval type

- It defines an interval of an ordered values set already defined from an ordinal type by indicating its lower and upper bounds.
- The type of both constants limiting the interval specifies the basic ordinal type of this interval.
- In Interval type, values are scalars and *Constant_1* must be greater than *Constant_2*.

Type *name_of_the_type=Constant_1 … Constant_2;*

Examples

Type

- Indices=1…10;
- Digit='0'…'9';
- Upper='A'… 'Z';
- Lower='a'… 'z';

Non-standard types (4)

Example: Let consider the following declarations:

```
Algorithm Non-Standard_Types;
Type
    Feast={Aid_El_Fitr, Aid_El_Adha, Moharram, Achoura, Mouloud};
    Months={January, February, March, April, May, June, July, August,
    September, October, November, December};
Variables f: Feast; m: Months;
```

```
Begin
     vacation=0;
     print("Give a month m="); read(m);
     If(m>=March) and (m<=June)
         print("It is Spring");
     For (f=Aid_El_Fitr; f<=Mouloud; f++)
         vacation=vacation+1;
End;
```

Non-standard types (5)

- It is possible to create a non-standard type from another one.

Example:

Type

Rainbow={Violet, Indigo, Blue, Green, Yellow, Red};

Year={September, October, November, December, January, February, March, April, May, June, July, August};

Color=Violet…Green;

Academic_Year= October…June;

Non-standard types (5)

```
Algorithm Final_Example_Chapter_Two;
Constant   PI = 3.14;  Hundred = 100;
Types
    Answer ={Yes, No};
    Feast={Aid_El_Fitr, Aid_El_Adha, Moharram, Achoura, Mouloud};
    Months={January, February, March, April, May, June, July, August,
    September, October, November, December};
    NumberMonths = 1…12;
    Temperature = -15 … 60;
    LowerCase = ‘a’ … ’f’;
Variables
    index, lowerIndex, higherIndex: Integer;   result, total: Real;
    num : NumberMonths;                vacation : Feast;
    resp : Answer;                     temp : Temperature;
    code1, code2 : LowerCase;          m1, m2, m3 : Months;
```

Thank You!

abdelfetah.hentout@enscs.edu.dz
hentout.abdelfetah@gmail.com